Overview

- DAGs again: An example
- Scheduling
  - Greedy
  - Work stealing
- Cilk
- Background material:
  - Blumofe, Leiserson, Scheduling Multithreaded Computations by Work Stealing, Journal ACM, 46(5), 1999

Example: Fibonacci Numbers

```c
int fib (int n) {
    if (n<2) return (n);
    else {
        int x,y;
        x = spawn fib(n-1); // can execute in parallel with parent
        y = fib(n-2);
        sync;
        return (x+y);
    }
}
```

Stupid way of computing (why?)
But good example

Graphs obtained this way are called nested parallel (or fully strict):
- Every thread has one incoming edge (the spawn edge)
- All join edges from a thread connected to the parent thread

Assuming every node has unit time:
\[ W = 17, D = 7 \]

How to Schedule on p Processors?

Node: Sequence of instructions without call, spawn, sync, return
Edge: Dependency

The DAG unfolds dynamically:
Greedy Scheduler

- **Idea:** Do as much as possible in every step
- **Definition:** A node is ready if all predecessors have been executed

**Complete step:**
- \( \geq p \) nodes are ready
- run any \( p \)

**Incomplete step:**
- \(< p \) nodes ready
- run all

**How good is this theoretically?**

(blackboard)

Greedy Scheduler: Sketch

Maintain thread pool of live threads, each is ready or not

- **Initial:** Root thread in thread pool, all processors idle
- **At the beginning of each step each processor is idle or has a thread \( T \) to work on**
  - **If idle**
    - Get ready thread from pool
  - **If has thread \( T \)**
    - Case 0: \( T \) has another instruction to execute
    - execute it
    - Case 1: thread \( T \) spawns thread \( S \)
      - return \( T \) to pool, continue with \( S \)
    - Case 2: \( T \) stalls
      - return \( T \) to pool, then idle
    - Case 3: \( T \) dies
      - if parent of \( T \) has no living children, continue with the parent, otherwise idle

Greedy Scheduler: Problems

- **Centralized**
- **Overhead**

**Work stealing scheduler:**

- thread pool distributed
- all processors do only useful work and operate locally as long as there is work to do
- Good asymptotic behavior, good practical behavior
- Implemented in Cilk runtime system
Work Stealing Scheduler

Each processor maintains a “ready deque” deque of threads ready for execution; bottom is manipulated as a stack.

- Threads can be removed or added (stack discipline).
- The thread being executed is highlighted.

Diagram:
- Each processor maintains a “ready deque” deque of threads ready for execution; bottom is manipulated as a stack.
- Threads can be removed or added (stack discipline).
- The thread being executed is highlighted.
Work Stealing Scheduler

- Each processor maintains a “ready deque:” deque of threads ready for execution; bottom is manipulated as a stack

- Steal from the top of a randomly selected processor

Work Stealing Scheduler: Sketch

Each processor maintains a ready deque, bottom treated as stack

- Initial: Root thread in deque of a random processor
- Deque not empty:
  - Processor takes thread T from bottom and starts working
  - T spawns S: Put T on stack, continue with S
  - T stalls: Take next thread from stack
  - T dies: Take next thread from stack
  - If T enables a stalled thread S, S is put on the stack of T’s processor
- Deque empty:
  - Steal thread from the top of a random (uniformly) processor’s deque
- Theoretical performance? (blackboard)

Cilk

- Extension of C/C++
- Compiler and runtime system
- Developed at MIT, now distributed by Intel
- Cilk home at Intel